連載 3

割り算回路設計，あのての手

CSAを用いた除算回路の高速化

鈴木昌治

デバイスの記事

関連データ

非回復法による除算回路の高速化を図るため，ここではCSA（carry save adder）を用いた手法を紹介する．この手法は乗算回路の高速化にも利用されている．除算回路に適用するには部分商の判定についてくふうする必要がある．なお，本稿で設計した回路のソース・コードは本誌ホームページ（http://www.cqpub.co.jp/dwm/）からダウンロードできる．

（編集部）

連載第1回（本誌2005年8月号，pp.109-117）では回復法を用いて除算の基本を押さえ，第2回（同年10月号，pp.136-144）では非回復法で符号付きの引き数の取り扱いへ発展させて，高速化へのアプローチの土台作りを行いました．今回は，ここまで説明してきた構造をいかに高速化するかについて検討してみましょう．

1 入出力特性──前回までのおさらい

図1に回復法と非回復法における，1サイクル当たりの入出力特性を示します．今回はこの図が重要な役割を担うので，最初に図の示す意味をしっかり理解しておきましょう．同時に，回復法と非回復法のおさらいもしておきます．

● 演算の条件を評価する際に利用しよう

まずは回復法から説明します．iサイクル目を開始するときの入力をpr_iとします．$i-1$サイクル目の部分余を1ビットだけシフト・アップしたものになります．除数をdとすると，回復法における除算成立条件は次式で表せます．

$$0 \leqq pr_i < 2d \quad \cdots\cdots (1)$$

式(1)を前提とし，部分商をpq_iとすると，iサイクル目では次のような操作を行います．

$$\left.\begin{array}{l} pq_i = 1 \\ pr_{i+1} = pr_i - d \end{array}\right\} (pr_i \geqq d\text{の場合}) \quad \cdots\cdots (2\text{-}1)$$

$$\left.\begin{array}{l} pq_i = 0 \\ pr_{i+1} = pr_i \end{array}\right\} (pr_i < d\text{の場合}) \quad \cdots\cdots (2\text{-}2)$$

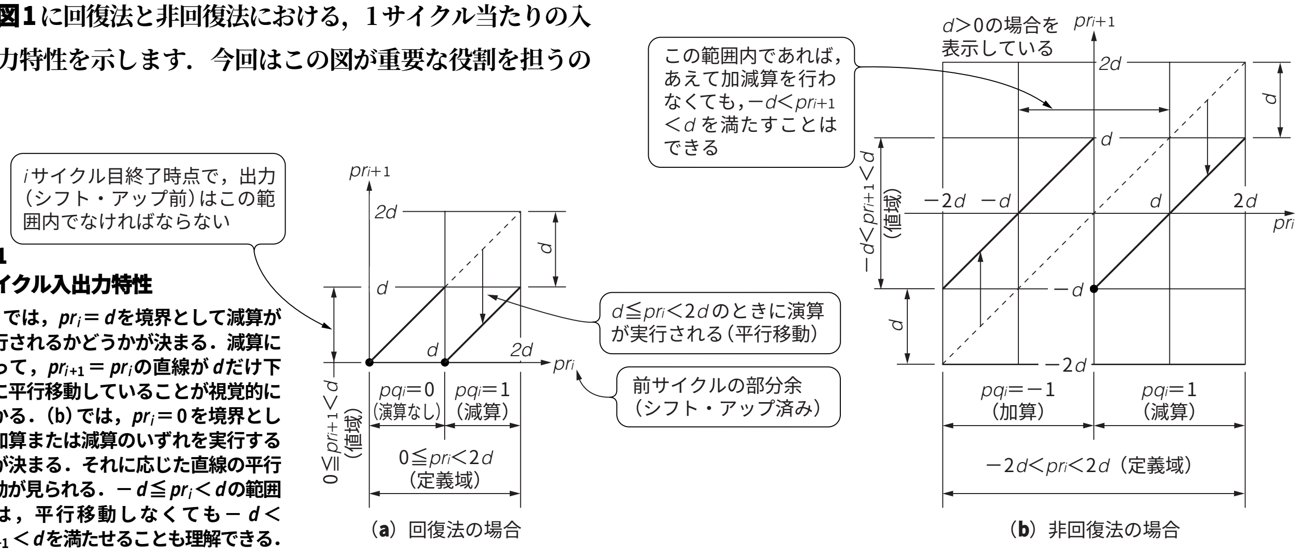


図1 サイクル入出力特性

(a)では，$pr_i = d$を境界として減算が実行されるかどうかが決まる．減算によって，$pr_{i+1} = pr_i$の直線がdだけ下方に平行移動していることが視覚的にわかる．(b)では，$pr_i = 0$を境界として加算または減算のいずれを実行するかが決まる．それに応じた直線の平行移動が見られる．$-d \leqq pr_i < d$の範囲では，平行移動しなくても$-d < pr_{i+1} < d$を満たせることも理解できる．

これに基づいて，**図1(a)**を見てみましょう．図において点線で示した部分は，出力＝入力とした(減算が実行されなかった)場合の特性，実線で示した部分は実際の入出力特性となります．

横軸は入力のpr_iであり，シフト・アップ済みなので式(1)の条件を満たしています．これが定義域となります．$pr_i \geqq d$の場合のみ除数の減算が実行され，部分商を1とします．このため，出力pq_{i+1}(縦軸)はdだけ下方向に平行移動した値をとります．この操作により，値域が$0 \leqq pr_{i+1} < d$の範囲に抑えられるのがよくわかると思います．

図1(b)に，非回復法の場合を示します．グラフのつごう上，$d > 0$としてあります．まず，除算成立条件を式(3)に，iサイクル目で行う操作を式(4-1)と式(4-2)に示します[注1]．

$$-2d < pr_i < 2d \quad \cdots\cdots (3)$$

$$\left.\begin{array}{l} pq_i = 1 \\ pr_{i+1} = pr_i - d \end{array}\right\} (pr_i \geqq 0 \text{の場合}) \quad \cdots\cdots (4\text{-}1)$$

$$\left.\begin{array}{l} pq_i = -1 \\ pr_{i+1} = pr_i + d \end{array}\right\} (pr_i < 0 \text{の場合}) \quad \cdots\cdots (4\text{-}2)$$

減算/加算が施された領域(実線)は，点線(演算が行われなかった部分)が$\pm d$だけ平行移動しているのがわかります．また，実線の端点における黒丸はその点が実在することを示しており，非回復法で$pr_i = 0$の場合は$pr_{i+1} = -d$となって，除算の成立条件である式(3)が満たされていないことがすぐにわかります．

このように，入出力の関係がひと目でわかるので，場合分けの条件や除算の成立条件などを評価するには，非常に便利な図です．

また，**図1(b)**から非回復法において注目すべき点がわかります．入力が$-2d \leqq pr_i < 2d$の場合，出力は$-d \leqq pr_{i+1} < d$の範囲で一意に決まればよいので，入力が$-d \leqq pr_i < d$の範囲にある場合は減算や加算は必要ないということです．言い換えれば，入力が$-d \leqq pr_i < d$の範囲にある場合は平行移動せず，点線の位置のままでも非回復法の条件は成り立つということです．

2 SRT法に見る回路高速化のヒント

次に，この発想を利用したおもしろい除算法を紹介しましょう．非回復法の変型ともいえる「SRT法」です．個人的には，ここで説明するSRT法は「実用性の面ではいまひとつ」と考えていますが，その中にある発想が後述の高速化の手法につながるので，ここで紹介しておきます．

● 加減算や比較演算を省略できるメリットはあるが…

まず，除数を$1 > d \geqq 1/2$に正規化し，$-1/2 \leqq pr_i < 1/2$になるように調整しますが，これらは符号反転とシフトで行えます．そして，**表1**の条件で部分商の選択と加減算を行うこととします．

この条件下で，先ほどの入出力特性をプロットしてみましょう．これを**図2**に示します．非回復法の場合と似た特性ですが，

- 加減算を行わない領域がある
- その境界が$\pm 1/2$という固定値

という点が異なります．また，入力が$-1 \leqq pr_i < 1$，出力が$-1/2 \leqq pr_i+1 < 1/2$で定義されることも，非回復法とは違います．

ここからわかることは，$-1/2 \leqq pr_i < 1/2$である限り$-1/2 \leqq pr_{i+1} < 1/2$も成立するので，最初のサイクルで条件が成立すれば，最後まで破たんなく除算が成立するということです．このアルゴリズムが持つメリットは，次の二

表1　SRT法の処理条件

pr_iの範囲	部分商	演　算
$pr_i < -1/2$	-1	$pr_{i+1} = pr_i + d$
$-1/2 \leqq pr_i < 1/2$	0	$pr_{i+1} = pr_i$
$1/2 \leqq pr_i$	1	$pr_{i+1} = pr_i - d$

注1：グラフのつごう上$d > 0$としたので，式(4-1)と式(4-2)では判断基準を$pr_i \geqq 0$としてあるが，dを符号付きで扱う場合はpr_iとdの符号が等しいか否かが判断基準となる．

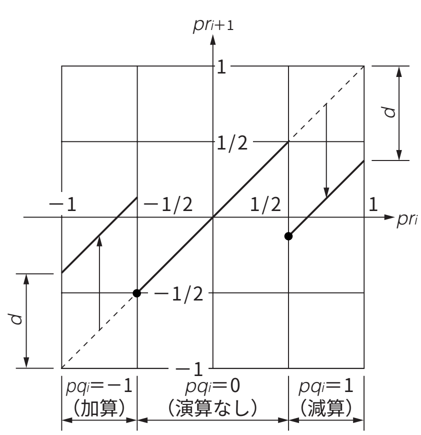


図2　SRT法におけるサイクル入出力特性
正規化された除数dに対し，±1/2という固定値を境界として加算/減算を使い分ける方式でも，演算しない領域を持つことで除算の成立条件を満たすことができる．

つです．

- 条件によっては加減算を行うことなく，入力をシフトしただけのものが出力となる．そのため，この処理を行うサブモジュールを縦続接続し，パイプライン・レジスタを入れないとすれば，処理にかかる平均速度が上がる
- **表1**の判定を行うにあたって，2のべき乗（±1/2）を固定の境界としているので，pr_iの上位2ビットだけで判定でき，dとの比較は必要ない

もちろん，同期回路全盛の昨今では，パイプライン化してしまえば，加減算をスキップしようがしまいが，何の得にもならないといえます．さらに，dとの比較なしに部分商を判定できたところで，加減算は必要な処理なので高速化には大きく貢献しません．こうした点から，筆者は「SRT法単独では実用的ではない」と考えるわけです．

ところで，非回復法のように処理の境界が$\pm d$であるというのは直感的に納得できると思いますが，なぜ±1/2で正しい除算が可能となるのでしょう．

これは，非回復法が$\pm d$という演算機能を備えているにもかかわらず，加算か減算かの二つの選択肢しかなかったのに対し，SRT法では「演算しない」という三つ目の選択肢を手に入れたことに起因します．$1 > d \geqq 1/2$の条件で±1/2を境界として判定したことによる誤差は1/2未満ですが，演算しない範囲を持つことでよりきめ細かい演算が可能になり，その誤差を抑え込んでいるといえます．かりに，**図2**で演算しない領域を持たず，非回復法と同様に$pr_i = 0$を境界として加減算を行ったとしましょう．$-1/2 \leqq pr_{i+1} < 1/2$が成立しないことは，図から明らかです．

実用性に乏しい（ということで，ブロック図やRTLソース・コードは省略）という点は残念なのですが，くふうしだいでおもしろいアルゴリズムができるというのは楽しいことですね．

3 CSAによる高速化

回復法では，減算結果の符号によって減算の可否を判定します．このため，この判定部が演算速度を決定する遅延パスになります．

一方，非回復法ではサイクル先頭で加算か減算かが決定できるような構造を適用してこの遅延パスを短縮します．

しかし，加減算そのものがなくなったわけではないので，加減算のパスがスピード面でのボトルネックであることに変わりありません．これは，SRT法においても同じです．また，4を法とする構造に持ち込んだとしても，ゲート規模が3倍以上になるにもかかわらず，速度は2倍以上には向上しません．サイクル単位でパイプライン化すれば，パイプライン間の遅延はむしろ大きくなる方向であることもわかっています．加減算のパス，ここを何とかしないかぎり，劇的な高速化が見込めないことが見えてきました．

● 乗算の高速化手段はそのまま除算に適用できるか？

「除算は乗算と表裏一体である」と本連載第1回で述べましたが，乗算における高速化の手段としてcarry save adder（CSA）が挙げられます．キャリの伝播を一時的に保留して加算の遅延をビット単位に分解する方法[注2]です．除算においても，キャリを下位から上位に順次伝播するcarry propagation adder（CPA）である必要はまったくありません．つまり，CSAを採用することによって加減算の遅延はフル・アダー1段分の遅延にできるわけです．除数のビット数が大きくなっても，それには関係なく，つねにフル・アダー1段分の遅延で加減算を終えることができます．

しかし，良いことばかりではありません．乗算が部分積の加算を単純に繰り返すことで最終的な積に到達できるのに対し，除算はサイクル単位で部分余によって部分商を判定したうえで繰り返しを実行する必要があります．CSAを採用することで，部分余がsum（和）とcarry（キャリ）の2値に分かれた形で保存されるので，部分余の判定が従来の手法ではできないことになります．といって，部分余の判定のために，sumとcarryを毎回加算し直すのでは，まさに本末転倒です．sumとcarryの全ビットを加算してみなければ判定できないとすれば，その加算器の遅延がCPAの場合と同じになり，CSAを採用した意味がありません．

では，ほんとうにsumとcarryの全ビットを加算してみなければ部分商は判定できないのでしょうか．実は，ここに一つの抜け道があるのです．

● sumとcarryの上位けたのみでおおまかな判定を行う

CSAによって実際に加減算するけたのさらに上位の限られたけたのみで部分商を判定できれば，そこにCPAを採用したとしても，まともに全ビットの加算を行うよりは小さな遅延で済むだろうということは，容易に想像できるでし

注2：つまり，下位からのキャリを待たずに，けたの合っている項を手当たりしだいに加算する．

ょう．さらに，CPA の代わりに判定まで含めたテーブルを採用すれば，遅延の面でさらに有利になるはずです．

まず，**図1(b)**を再度ご覧ください．ここでは$d > 0$として話を進めますが，すでに説明したように入力が$-d < pr_i < d$の範囲にある場合は，あえて加減算を実行しなくても出力を$-d < pr_{i+1} < d$の範囲に収められます．

非回復法では加減算を実行しない範囲を持っていないので**図1(b)**のようになっているわけですが，SRT 法で採用した加減算を実行しない範囲を適用してもかまいません．この「かまわない」というところが重要で，「かまわない＝どちらでもよい」と解釈すれば，**表2**の処理に変形することができます．これが「およその判定」となります．sum と carry の上位のみを加算したpr_iの近似値で部分商を判定してしまうと，当然，判定には切り捨てによる誤差が生じることになります．この判定誤差を「どちらでもよい部分（冗長な領域）」で吸収してしまうのです．

ところで，非回復法において途中のサイクルで完全に割り切れてしまった場合に演算エラーを起こすため，後処理によって補正するということを前回お話ししました．これは$pr_i = 0$のときでも加減算を実行してしまうことに問題があるわけですが，今回は「加減算を実行しない」機能を適用しようと考えているため，$pr_i = 0$のときには加減算を実行しないように，部分商を判定する時点で考慮してあります．この点も**表2**には盛り込んであります（いずれテーブル化されるとすれば，きめ細かい条件づけを行っても遅延には影響しない）．**表2**の条件におけるサイクルの入出力特性を**図3**に示します．図から除算の成立条件が満たされていることは明白です．

● 除数 d を判定条件から外す

さて，「どちらでもよい」という条件のままでは回路を作ることができません．どのようにアルゴリズムを組み立てるかを考えましょう．ここで，pr_iは sum と carry で保存された部分余の和である真の部分余とします．サイクル当たりの sum と carry をそれぞれps_i，pc_iと定義すると，式(5)が成立します．

$$pr_{i+1} = ps_i + pc_i \quad \cdots\cdots(5)$$

では，ps_iとpc_iからいかにしてpr_{i+1}を推定すればよいのでしょうか．ps_iとpc_iの上位けたを用いることはすでに説明したとおりですが，遅延の小さなCPA で済ませたい，あるいはテーブル化してしまいたいということになると，ビット幅はかなり限定されます．さらに，ps_i，pc_iの加算結果とdの大小関係を検証していたのではCSA に至るまでの手順が長くなり，遅延削減の効果が半減してしまいます．できればdとは無関係にps_i，pc_iだけで判定を行いたいところです．そこで，除数を式(6)のように正規化して値の範囲を限定し，部分商の判定からdを削除してしまいます（ここで，SRT 法の発想が生かされる）．

$$1 > d \geqq 1/2 \quad \cdots\cdots(6)$$

こうしたうえで，ps_iとpc_iの上位4ビットを判定に用いることにします．ps_iとpc_iの上位4ビットをそれぞれps_spq_i，pc_spq_i，2値の和をsm_spq_iとします．すると，式(7)が成立します．

$$sm_spq_i = ps_spq_i + pc_spq_i \quad \cdots\cdots(7)$$

dの値の範囲に合わせて，ps_spq_iとpc_spq_iのデータ・フォーマットは**図4**となります．小数以下が2ビットなので

表2　CSA 採用による非回復法の処理条件

pr_iの範囲	部分商	演　算
$pr_i \leqq -d$	-1	$pr_{i+1} = pr_i + d$
$-d < pr_i < 0$	-1 または 0	$pr_{i+1} = pr_i + d$ または $pr_{i+1} = pr_i$
$pr_i = 0$	0	$pr_{i+1} = pr_i$
$0 < pr_i < d$	0 または 1	$pr_{i+1} = pr_i$ または $pr_{i+1} = pr_i - d$
$d \leqq pr_i$	1	$pr_{i+1} = pr_i - d$

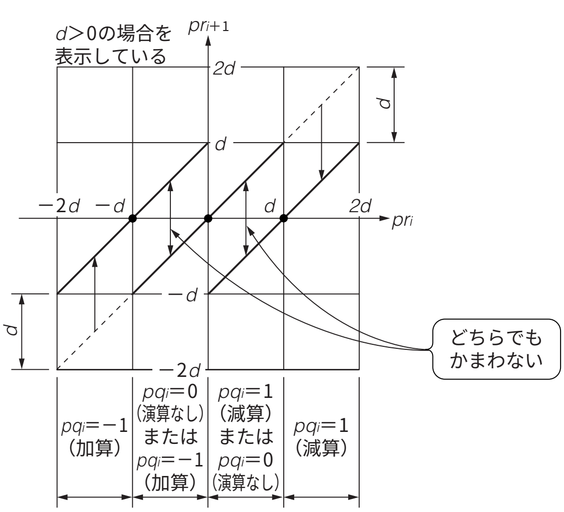


図3
CSA を採用した非回復法におけるサイクル入出力特性
「$-d \leqq pr_i < d$の範囲では加減算を実行してもしなくてもよい」とすることで，判定誤差を吸収する．

分解能は0.25となり，sm_spq_iの真の部分余に対する誤差は0.5未満となります．

pr_iの代用としてsm_spq_iを用いるという条件のもとで**図3**を書き直すと，**図5**のようになります．上位4ビットに丸められた（切り捨てられた）ことによる誤差を考慮すると，真値の存在範囲は図中の灰色の領域になります．ここから式(1)を満たして除算が成立するように判定していけばよいわけです．結果的には**図5**に示したとおりに判定することで，除算を成立させることができます（分解能が0.25なので，図には直線ではなく点をプロットしてある）．

図5の例では，$sm_spq_i = -0.5$の場合，本来は加算を実行しなくてもよいのですが，$d = 0.5$の場合に$pr_{i+1} = -d$となってしまうため，加算を行う範囲としました．また，**図5**において$sm_spq_i = 1$の場合，誤差の範囲を考えると減算を実行しても$pr_{i+1} > d$となってしまう可能性がありそうです．しかし，式(1)の条件の入力を真値としていれば，

$$pr_{i+1} = pr_i - d < 2d - d = d \quad \cdots\cdots(8)$$

となり，式(1)の条件は満たされ続けます．

以上の判定条件を，**表3**にまとめます．

このようにして，dを判定条件に入れず（ただし，dは式(6)のように正規化されていること），sumとcarryの上位4ビットで部分商の判定が可能となりました．したがって，4ビットの二つの数，合計8ビット入力のテーブルにより，すべての判定ができることになります．テーブルのサイズとしては十分現実味がある値なので，sumとcarryを加算して結果をデコードするよりは高速化が期待できます．

ところで，前回の非回復法では，除算後に商と余に補正が必要でした．今回のケースでも，CSAを採用しても基本的なアルゴリズムは非回復法のままなので，やはり補正が必要となります．ただし，次の点が大きく異なります．

- $d > 0$として処理しているので，dの符号について考える必要がない
- 式(1)を満たすように考慮してきたので，$pr_{i+1} = \pm d$の場合については考える必要がない

こうした点を踏まえた後処理の補正を**表4**に示します．

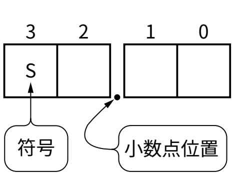


図4　ps_i，pc_iのデータ・フォーマット

小数点以下は2ビットなので，分解能は0.25となる．

表4　余と商の補正

r	z	余補正	商補正
正	正	補正なし	補正なし
	負	$-d$	$+1$
0	―	補正なし	補正なし
負	正	$+d$	-1
	負	補正なし	補正なし

表3　CSAを採用した非回復法の処理条件（上位4ビットによる判定）

sm_spq_i	部分商	演　算
1000～1110	-1	$pr_{i+1} = pr_i + d$
1111～0000	0	$pr_{i+1} = pr_i$
0001～0111	1	$pr_{i+1} = pr_i - d$

4　CSAによる非回復法除算の回路設計

それでは，CSAを採用した非回復法除算回路を設計してみましょう．被除数を16ビット，除数を8ビットの符号付き整数とします．$-2d < pr_{initial} < 2d$とするには，前回までと同じく回路を簡略化するため，被除数の上位に符号を拡張することにします．除数は符号を正とし，式(6)のとおり正規化されているものとします．

● ループ動作の遅延をいかに小さく抑えるかがポイント

図6にブロック図を示します．前処理（初期化），ループ処理，後処理と3段階に分けて図示しています．なお，シーケンス部分は示していません．

この回路の動作は，以下のようになります．

1) re（リセット・イネーブル）でシーケンスを初期化する（$i = 0$）．

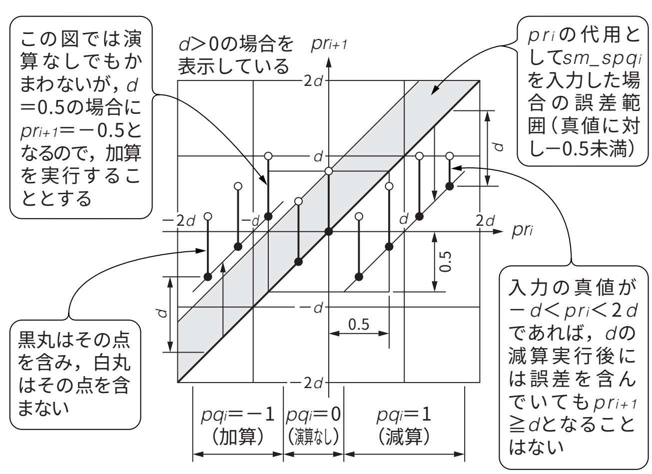


図5　CSAを採用した非回復法におけるサイクル入出力特性（上位4けた）

上位けたに限定された入力で真値を推定しての判定，演算となるので，特性は直線ではなく幅を持った領域で表現される．また，限定されたけたにより分解能が低くなるので，離散的な表現となる．

2）データ格納用レジスタは3系統存在する．drは除数dの，srはCSA出力のsumの，crはcarryの格納用レジスタである．srには初期状態として被除数zを格納する．そのとき，上位に符号拡張を行うために7ビット分余分にけたをとるが，sumを格納することと，後処理でzの符号が必要となることから，合計10ビット上位に拡張する．crもcarryの格納を行うので，9ビット上位に拡張しておく．

3）startパルスでzはsrに，dはdrに格納され，crはcarryなしということで0にリセットされて，除算

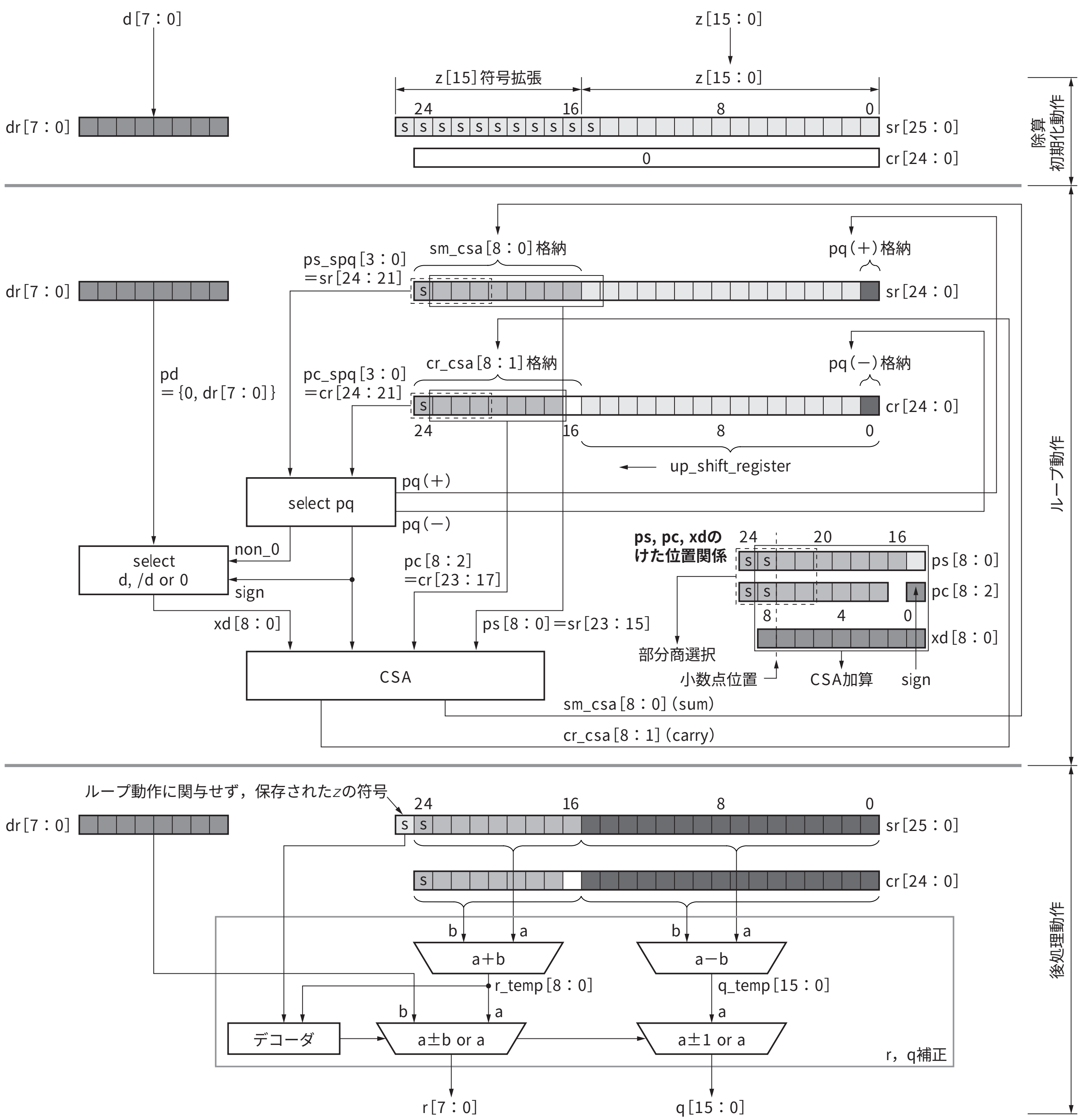


図6　CSAを採用した非回復法除算ブロック

drは除数dを，srはCSA出力の和を，crはキャリをそれぞれ格納するためのレジスタ．srには初期状態として被除数zを格納する．ループ動作においては，srとcrの上位けたで部分商pq(＋)，pq(－)を判定し，CSAで加算されるxdを生成する．ps，pc，xdがCSAで加算され，結果を格納レジスタに返す．

の初期化が終わる．$i = 16$にセットされる．

4) 前サイクルの*ps_spq*（sum上位4ビット）と*pc_spq*（carry上位4ビット）を用いて，select pq（部分商判定ブロック）において*pq*（部分商）が決まる．pq(＋)は部分商が＋1のときに1，pq(－)は部分商が－1のときに1となり，それ以外では0とする．同時にselect pqでは，CSAで使用する*xd*（$\pm d$または0）を生成するための制御信号non_0（sm_spq≠0を表す），sign（*sm_spq*の符号）を生成する．これにより，*xd*を生成して，CSAで加算される．$-d$の代わりにdを反転するのみで，$-d$とするための＋1はsignをCSAで加算することで実現する．このようにして，CSAでは$ps + pc + xd$が演算されてこのサイクルのsum（*sm_csa*），carry（*cr_csa*）が決定する．次のサイクルに移行するにあたり，pq(＋)はsrのLSBに，pq(－)はcrのLSB

リスト1　図6の回路構成のRTL記述

```
module        div_nonrstr_csa_a  (
  q     ,  // quotient  [15:00]:signed integer
  r     ,  // remainder [07:00]:signed integer
  busy  ,  // busy

  z     ,  // dividend [15:00]:signed integer
  d     ,  // divisor  [08:00]:unsigned integer
  start ,  // start (divider trigger)
  clk   ,  // clock
  re   );  // reset ena

// io
  input [15:00]  z    ;
  input [07:00]  d    ;
  input start         ;
  input clk           ;
  input          re   ;

  output [15:00] q    ;
  output [08:00] r    ;
  output         busy ;

// reg
  reg    [04:00] i    ;
  reg    [25:00] sr   ;
  reg    [24:00] cr   ;
  reg    [07:00] dr   ;

// sequence (シーケンサ)
  wire  en_seq = ( i > 0 ) ;

  always @( posedge clk ) begin
        if ( re )          i <= 5'h00    ;
        else if ( start )  i <= 5'h10    ;
        else if ( en_seq ) i <= i - 1'b1 ;
     end

// function (演算処理部)
  // select pq (sumとcarryの上位4ビット加算によるpqの判定)
  wire  [03:00] ps_spq = sr[24:21]       ;
  wire  [03:00] pc_spq = cr[24:21]       ;
  wire  [03:00] sm_spq = ps_spq + pc_spq ;
  wire          pq_p   = ~sm_spq[03] & |sm_spq[02:00] ;
                                                //pq(+)
  wire          pq_n   = sm_spq[03] & ~&sm_spq[02:00] ;
                                                //pq(-)
  wire          sn_sd  = ~sm_spq[03]                  ;
                                                 //sign
  wire          n0_sd  = pq_p | pq_n                  ;
                                                //non_0

  // select d (pqの判定結果により加減算項xdを選択)
  wire  [08:00] pd = {1'h0,dr[07:00]} ;
  wire  [08:00] xd = ( ~n0_sd )? 9'h000 : ( sn_sd )?
                                            ~pd : pd ;

  // carry save adder  (CSAはFull_Adderアレーで表現)
  wire  [08:00] ps = sr[23:15] ;
  wire  [08:00] pc = cr[23:15] ;
  wire  [08:00] sm_csa ;              // sum   CSA
  wire  [08:00] cr_csa ;              // carry CSA
  assign cr_csa[00] = 1'b0 ;
  assign {cr_csa[01],sm_csa[00]} = ps[00] + pq_p +
                          xd[00] ; // full adder array
  assign {cr_csa[02],sm_csa[01]} = ps[01] + 1'b0 +
                                              xd[01] ;
  assign {cr_csa[03],sm_csa[02]} = ps[02] + pc[02] +
                                              xd[02] ;
  assign {cr_csa[04],sm_csa[03]} = ps[03] + pc[03] +
                                              xd[03] ;
  assign {cr_csa[05],sm_csa[04]} = ps[04] + pc[04] +
                                              xd[04] ;
  assign {cr_csa[06],sm_csa[05]} = ps[05] + pc[05] +
                                              xd[05] ;
  assign {cr_csa[07],sm_csa[06]} = ps[06] + pc[06] +
                                              xd[06] ;
  assign {cr_csa[08],sm_csa[07]} = ps[07] + pc[07] +
                                              xd[07] ;
  assign              sm_csa[08] = ps[08] + pc[08] +
                                              xd[08] ;

  // next reg contents (次のサイクルのためのレジスタ内容)
  wire  [25:00] sw = {sr[25],sm_csa[08:00],sr[14:00],
                                             pq_p} ;
  wire  [24:00] cw = {cr_csa[08:00],cr[14:00],pq_n} ;

  always @( posedge clk ) begin
        if ( start ) begin
            sr <= {{10{z[15]}},z} ;
            cr <= 25'h0000000     ;
        end
        else if ( en_seq ) begin
            sr <= sw ;
            cr <= cw ;
        end
  end

  always @( posedge clk ) begin
        if ( start ) dr <= d ;
  end

  // final compensation (後処理補正)
  wire  [15:00] q_temp = sr[15:00] - cr[15:00] ;
  wire  [08:00] r_temp = sr[24:16] + cr[24:16] ;
  wire          r_eq0  = ~|r_temp ;
  wire  [15:00] q_cmpe = ( r_eq0 ) ? q_temp :
                         ( sr[25] )?
                         ( r_temp[08] )? q_temp
                                      :q_temp + 1'b1 :
                         ( r_temp[08] )? q_temp - 1'b1
                                       :q_temp        ;
  wire  [08:00] r_cmpe = ( r_eq0 ) ? r_temp :
                         ( sr[25] )?
                         ( r_temp[08] )? r_temp      :
                                         r_temp - dr :
                         ( r_temp[08] )? r_temp + dr :
                                         r_temp      ;

  assign        q      = q_cmpe ;
  assign        r      = r_cmpe ;
  assign        busy   = en_seq ;

endmodule
```

に，*sm_csa*（9ビット）はsr［24:16］に，*cr_csa*（8ビット）はcr［24:17］に戻される．さらに，sr［14:0］に格納されていた未使用の*z*は1ビットだけシフト・アップして，sr［14］は次のpsのLSBとなる（**図6**の*ps*，*pc*，*xd*のけた位置関係を参照）．$i=i-1$が実行される．

5）4）の処理を繰り返すことで，sr上の*z*のデータは上位から1ビットずつ消化され，sr上から消えていく．代わりに，srのLSBに順次格納されたpq（＋）が*z*のシフト・アップといっしょに上位へシフトし，srの下位側に蓄積される．同じく，crにはpq（−）が蓄積される．

6）$i=0$となったところでループ動作終了．sr［23:16］とcr［23:16］の和が余（r_temp）となり，sr［15:0］とcr［15:0］の差が商（q_temp）となって，*z*はすべてレジスタ上からなくなる．

7）後処理として，*z*の符号（z［25］）とr_tempの符号を比較し，商と余に**表4**による補正をかけて，除算終了．

この回路構成によるVerilog HDLのRTL記述を**リスト1**に示します．もっとも注意するべき点は，ループ部分の遅延をいかに小さくするかです．select pqにテーブルを採用することで4ビット加算とデコードの合成遅延を非同期メモリ1段分の遅延に集約すること，$-d$の実現をCSAと共同で行うことなど，特徴的なことがいくつかありますが，ターゲットとするデバイスによっては効果的ではない場合もありうるでしょう．

そもそも，米国Altera社や米国Xilinx社のFPGAに見られるようなCPAを前提として加算を行う構造には，CSAそのものが適合しにくいのです．こうしたFPGAでは，CPAでかなり高速な加算が行えるように最初から考慮されているので，CSAを適用して劇的な遅延の向上を期待するには無理があると言えます．除数のビット幅が大きい場合は効果を期待できますが，かけた手間に見合うかどうかは判断の難しいところです．ターゲット・デバイスに応じて，適当な選択が必要となります．

また，遅延をぎりぎりまで抑えようとしてCSAを採用しているので，ちょっとしたことで意図しない遅延を載せてしまっては意味がなくなってしまいます．むだを排除するように注意して設計を進めなくてはなりません．その意味では，**リスト1**はしくみを読みやすくする目的で書いているので，適当ではないことに注意してください．

● 除数*d*を符号付きで扱うには

ここまでは除数*d*を正の数として扱いましたが，実は*d*が符号付きであっても不つごうなことは何もないのです．単に，**図3**と**図5**を説明するつごう上，ひとまず符号なしということにしただけです．

まず，*d*が符号なしであった場合，$\pm d$を表現するために上位に1ビット拡張して，そこを符号ビットとしました．このため，*xd*を9ビットで扱いました．しかし，*d*が8ビット符号付きであれば，拡張する必要はありません．sumとcarryは，これに合わせてけたのとりかたを変更することになります（**図7**）．

以上の点を踏まえて，select pqの条件を**表5**のように変更します．*d*の符号が条件に加わります．また，後処理における商と余の補正にも変更が入ります．やはり，*d*の符号が条件に加わることになります（**表6**）．

大きな変更はこれだけです．ただし，前述のけたのとりかたが全体的に変わってきます．**図7**のブロック図，および**リスト2**のRTL記述で確認してみてください．動作としては，*d*が符号なしの場合とまったく同じです．

＊　＊　＊

今回はCSAを採用した非回復法を紹介しましたが，これを4を法とする回路に発展させたものが米国Intel社のPentiumプロセッサに使用されたそうです．構造的には今回説明したものに非常に近いものです（ただし，アルゴリ

◀ **表5　CSAを採用した非回復法の処理条件（*d*は符号付き）**

dの符号	sm_spq_i	部分商	演　算
正	1000～1110	−1	$pr_{i+1}=pr_i+d$
	1111～0000	0	$pr_{i+1}=pr_i$
	0001～0111	1	$pr_{i+1}=pr_i-d$
負	1000～1110	1	$pr_{i+1}=pr_i-d$
	1111～0000	0	$pr_{i+1}=pr_i$
	0001～0111	−1	$pr_{i+1}=pr_i+d$

▶ **表6　余と商の補正（*d*は符号付き）**

r	*z*	*d*	余補正	商補正
正	正	―	補正なし	補正なし
	負	正	$-d$	＋1
		負	$+d$	−1
0	―	―	補正なし	補正なし
負	正	正	$+d$	−1
		負	$-d$	＋1
	負	―	補正なし	補正なし

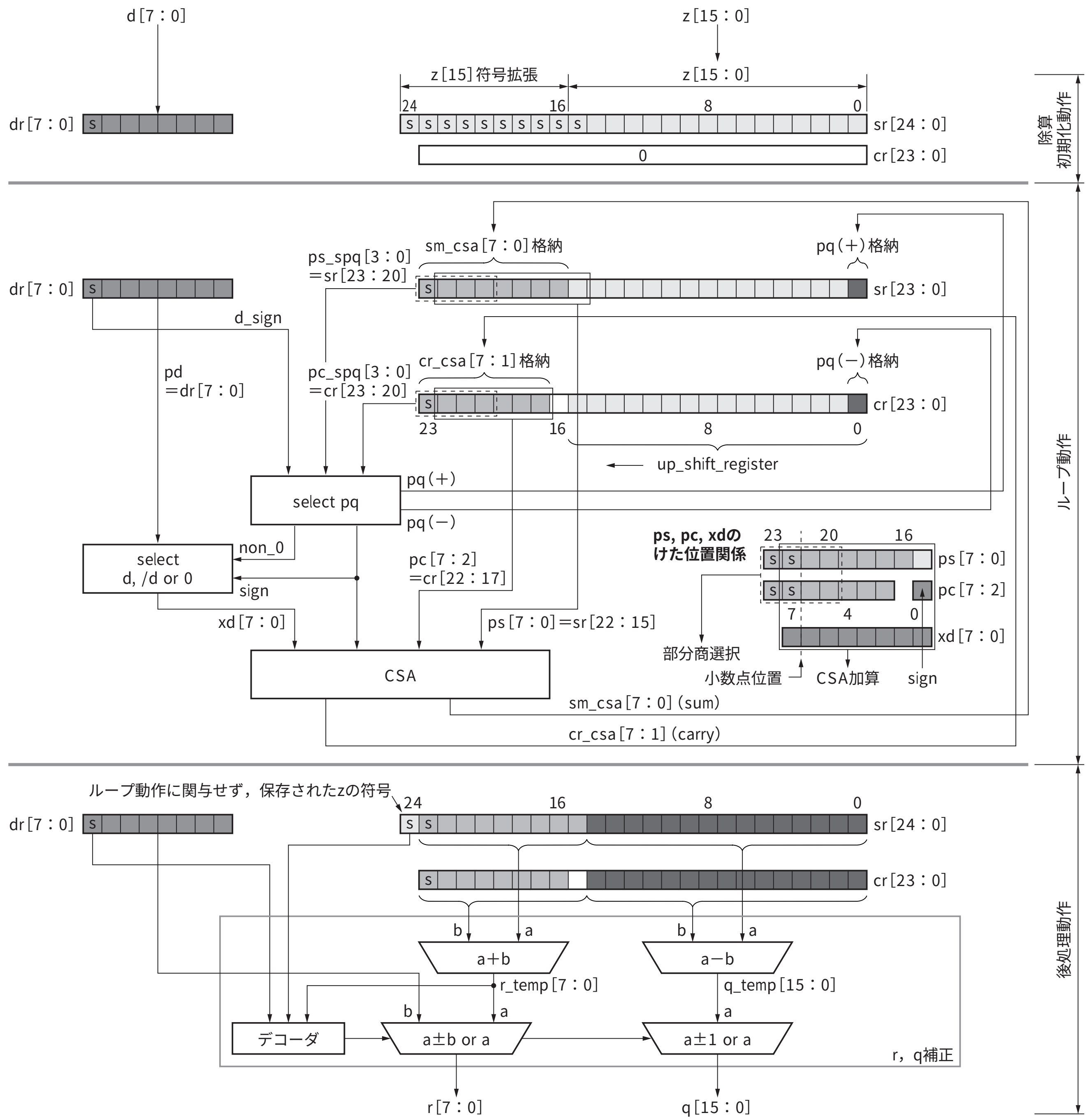


図7　CSAを採用した非回復法除算ブロック図（*d*は符号付き）
select pqの条件，および後処理における商と余の補正条件に*d*の符号が加わる．また，*d*のMSBが符号となったことで，けたのとりかたが多少変更する．

ズムはかなり複雑なので，詳細は割愛）．初期のPentiumプロセッサというと，除算のバグが印象に残るところですが，部分商を決定するテーブル（select pqの部分）にトラブルがあったと聞いています．除算というと枯れた回路と思っている方には，新鮮に聞こえるかもしれませんね（とはいっても，もうひと昔前の話になってしまったが…）．

すずき・しょうじ
スタンダード・ロジック（株）

<筆者プロフィール>
鈴木昌治．8月末に，「全日本マウンテンサイクリングin乗鞍」（やはり，ひたすら登るだけの自転車レース）に参加．春からの練習のかいがあって，昨年より6分ほど速く，自己ベスト．でも，昨年が遅すぎただけだったりして….

リスト2　図7の回路構成のRTL記述

```
module          div_nonrstr_csa_b  (
  q     , // quotient  [15:00]:signed integer
  r     , // remainder [07:00]:signed integer
  busy  , // busy

  z     , // dividend [15:00]:signed integer
  d     , // divisor  [07:00]:signed integer
  start , // start (divider trigger)
  clk   , // clock
  re   ); // reset ena

// io
  input [15:00]  z     ;
  input [07:00]  d     ;
  input          start ;
  input          clk   ;
  input          re    ;

  output [15:00] q     ;
  output [07:00] r     ;
  output         busy  ;

// reg
  reg [04:00]    i     ;
  reg [24:00]    sr    ;
  reg [23:00]    cr    ;
  reg [07:00]    dr    ;

// sequence（シーケンサ）
  wire  en_seq = ( i > 0 ) ;

  always @( posedge clk ) begin
        if      ( re )      i <= 5'h00    ;
        else if ( start )   i <= 5'h10    ;
        else if ( en_seq ) i <= i - 1'b1 ;
  end

// function（演算処理部．dが符号付きとなったことで複雑になる）
  // select pq（sumとcarryの上位4ビット加算によるpqの判定）
  wire      [03:00] ps_spq = sr[23:20] ;
  wire      [03:00] pc_spq = cr[23:20] ;
  wire      [03:00] sm_spq = ps_spq + pc_spq ;
  function [03:00] fsel_pq; // {pq(+),pq(-),sign,non_0}
        input d_sign          ;
        input [03:00] sm_spq ;
        casex ( {d_sign,sm_spq} )
              5'b0_10xx : fsel_pq = 4'b01_01 ;
              5'b0_110x : fsel_pq = 4'b01_01 ;
              5'b0_1110 : fsel_pq = 4'b01_01 ;
              5'b0_1111 : fsel_pq = 4'b00_00 ;
              5'b0_0000 : fsel_pq = 4'b00_10 ;
              5'b0_0001 : fsel_pq = 4'b10_11 ;
              5'b0_001x : fsel_pq = 4'b10_11 ;
              5'b0_01xx : fsel_pq = 4'b10_11 ;
              5'b1_10xx : fsel_pq = 4'b10_11 ;
              5'b1_110x : fsel_pq = 4'b10_11 ;
              5'b1_1110 : fsel_pq = 4'b10_11 ;
              5'b1_1111 : fsel_pq = 4'b00_10 ;
              5'b1_0000 : fsel_pq = 4'b00_00 ;
              5'b1_0001 : fsel_pq = 4'b01_01 ;
              5'b1_001x : fsel_pq = 4'b01_01 ;
              5'b1_01xx : fsel_pq = 4'b01_01 ;
        endcase
  endfunction
  wire [03:00] sel_pq = fsel_pq (dr[07] , sm_spq) ;
  wire         pq_p   = sel_pq[03] ; // pq(+)
  wire         pq_n   = sel_pq[02] ; // pq(-)
  wire         sn_sd  = sel_pq[01] ; // sign
  wire         n0_sd  = sel_pq[00] ; // non_0

  // select d（pqの判定結果により加減算項xdを選択）
  wire [07:00] pd = dr[07:00] ;
  wire [07:00] xd = ( ~n0_sd )? 8'h00 : ( sn_sd )?
                                                   ~pd : pd ;

  // carry save adder（CSAはフル・アダー・アレーで表現）
  wire [07:00] ps = sr[22:15] ;
  wire [07:00] pc = cr[22:15] ;
  wire [07:00] sm_csa         ; // sum CSA
  wire [07:00] cr_csa         ; // carry CSA
  assign       cr_csa[00] = 1'b0 ;
  assign      {cr_csa[01],sm_csa[00]} = ps[00] + pq_p
                         + xd[00] ; //フル・アダー・アレー
  assign      {cr_csa[02],sm_csa[01]} = ps[01] + 1'b0
                         + xd[01] ;
  assign      {cr_csa[03],sm_csa[02]} = ps[02] + pc[02]
                                     + xd[02] ;
  assign      {cr_csa[04],sm_csa[03]} = ps[03] + pc[03]
                                     + xd[03] ;
  assign      {cr_csa[05],sm_csa[04]} = ps[04] + pc[04]
                                     + xd[04] ;
  assign      {cr_csa[06],sm_csa[05]} = ps[05] + pc[05]
                                     + xd[05] ;
  assign      {cr_csa[07],sm_csa[06]} = ps[06] + pc[06]
                                     + xd[06] ;
  assign                  sm_csa[07]  = ps[07] + pc[07]
                                     + xd[07] ;

  // next reg contents（次のサイクルのためのレジスタ内容）
  wire [24:00] sw = {sr[24],sm_csa[07:00],sr[14:00],
                                                 pq_p} ;
  wire [23:00] cw = {cr_csa[07:00],cr[14:00],pq_n} ;

  always @( posedge clk ) begin
        if ( start ) begin
            sr <= {{9{z[15]}},z} ;
            cr <= 24'h0000000   ;
        end
        else if ( en_seq ) begin
            sr <= sw ;
            cr <= cw ;
        end
  end

  always @( posedge clk ) begin
        if ( start ) dr <= d ;
  end

  // final compensation
    （後処理補正．ここもdが符号付きとなるので複雑になる）
  wire [15:00] q_temp = sr[15:00] - cr[15:00] ;
  wire [07:00] r_temp = sr[23:16] + cr[23:16] ;
  wire         r_non0 = ( r_temp!=8'h0 )      ;

  function    [23:00] fcompe ;
        input [15:00] q_temp ;
        input [07:00] r_temp ;
        input [07:00] d      ;
        input         z_sign ;
        input         r_non0 ;
        casex ({r_non0 ,z_sign ,r_temp[07] ,d[07] })
            4'b0xxx : begin
                        fcompe[23:16] = r_temp ;
                        fcompe[15:00] = q_temp ;
                      end
            4'b100x : begin
                        fcompe[23:16] = r_temp ;
                        fcompe[15:00] = q_temp ;
                      end
            4'b1010 : begin
                        fcompe[23:16] = r_temp + d    ;
                        fcompe[15:00] = q_temp - 1'b1 ;
                      end
            4'b1011 : begin
                        fcompe[23:16] = r_temp - d    ;
                        fcompe[15:00] = q_temp + 1'b1 ;
                      end
            4'b1100 : begin
                        fcompe[23:16] = r_temp - d    ;
                        fcompe[15:00] = q_temp + 1'b1 ;
                      end
            4'b1101 : begin
                        fcompe[23:16] = r_temp + d    ;
                        fcompe[15:00] = q_temp - 1'b1 ;
                      end
            4'b111x : begin
                        fcompe[23:16] = r_temp ;
                        fcompe[15:00] = q_temp ;
                      end
        endcase
  endfunction

  wire [23:00] compe = fcompe (q_temp ,r_temp ,dr ,
                                     sr[24] ,r_non0) ;

  assign q    = compe[15:00] ;
  assign r    = compe[23:16] ;
  assign busy = en_seq       ;

endmodule
```